CSCI-GA 2590: Natural Language Processing

Predicting Sequences

Parijat Parimal
pp2206

Collaborators: None
By turning in this assignment, I agree by the honor code of the College of Arts and Science at New York University and declare that all of this is my own work.

Before you get started, please read the Submission section thoroughly.

Submission

Submission is done on Gradescope.

Written: When submitting the written parts, make sure to select **all** the pages that contain part of your answer for that problem, or else you will not get credit. You can either directly type your solution between the `shaded` environments in the released `.tex` file, or write your solution using pen or stylus. A `.pdf` file must be submitted.
Programming: Questions marked with "coding" next to the assigned to the points require a coding part in `submission.py`. Submit `submission.py` and we will run an autograder on Gradescope. You can use functions in `util.py`. However, please do not import additional libraries (e.g. `numpy`, `sklearn`) that aren't mentioned in the assignment, otherwise the grader may crash and no credit will be given. You can run `test.py` to test your code but you don't need to submit it.

Problem 1: N-gram Language Models

In this problem, we will derive the MLE solution of n-gram language models. Recall that in n-gram language models, we assume that a token only depends on $n-1$ previous tokens, namely:

$$p(x_{1:m}) = \prod_{i=1}^{m} p(x_i \mid x_{i-n+1:i-1}) \ ,$$

where $x_i \in \mathcal{V}$ and $x_{1:i}$ denotes a sequence of i tokens $x_1, x_2, \ldots, x_i$. Note that we assume all sequences are prepended with a special start token $*$ and appended with the stop token `STOP`, thus $x_i = *$ if $i < 1$ and $x_m =$ `STOP`. We model the conditional distribution $p(x_i \mid x_{i-n+1:i-1})$ by a categorical distribution with parameters α:

$$p(w \mid c) = \alpha[w, c] \quad \text{for } w \in \mathcal{V}, c \in \mathcal{V}^{n-1} \ .$$

Let $D = \{x^i_{1:m_i}\}_{i=1}^N$ be our training set of N sequences, each of length m_i.

1. [2 points] Write down the MLE objective for the n-gram model defined above. Note that we need to add the constraint that the conditional probabilities sum to one given each context.]

Ans:
Since, for $p(w \mid c) = \alpha[w, c]$ for

$$p(x_i \mid x_{i-n+1:i-1}) = (p(w \mid c)) = \alpha[w, c]$$

For N sequences distributed categorically with parameter α,

$$MLE = max\ \ell(\alpha) = max \sum_{i=1}^{N} \log \prod_{w \in \mathcal{V}; c \in \mathcal{V}^{n-1}} (\alpha[w, c])$$

$$= max \sum_{i=1}^{N} \sum_{w \in \mathcal{V}; c \in \mathcal{V}^{n-1}} \log(\alpha[w, c])$$

Thus, MLE can be given as the below expression along with the constraint mentioned subsequently.

$$MLE = max\ \ell(\alpha) = max \sum_{i=1}^{N} \sum_{w \in \mathcal{V}; c \in \mathcal{V}^{n-1}} \log(\alpha[w, c])$$

$$S.T. : \sum_{w \in \mathcal{V}; c \in \mathcal{V}^{n-1}} \alpha[w, c] = 1$$

2. [2 points] Write down the Langrangian $\mathcal{L}(\alpha, \lambda)$ using the method of Lagrange multipliers.

Ans:
From above, we can see that MLE has a contraint. We can rewrite the constraint as below:

$$g(\alpha) = \sum_{w \in \mathcal{V}; c \in \mathcal{V}^{n-1}} \alpha[w, c] - 1 = 0$$

Now, we apply Lagrangian on $\ell(\alpha)$ with constraint $g(\alpha)$ to get,

$$\mathcal{L}(\alpha, \lambda) = \ell(\alpha) - \lambda\ g(\alpha)$$

$$= \sum_{i=1}^{N} \sum_{w \in \mathcal{V}; c \in \mathcal{V}^{n-1}} \log(\alpha[w, c]) - \lambda(\sum_{w \in \mathcal{V}; c \in \mathcal{V}^{n-1}} \alpha[w, c] - 1)$$

Hence, we have,

$$\mathcal{L}(\alpha, \lambda) = \sum_{i=1}^{N} \sum_{w \in \mathcal{V}; c \in \mathcal{V}^{n-1}} \log(\alpha[w, c]) - \lambda(\sum_{w \in \mathcal{V}; c \in \mathcal{V}^{n-1}} \alpha[w, c] - 1)$$

3. [4 points] Find the solution for α. Define count$(\cdot)$ to be a function which maps a sequence to its frequency in D. You can assume count$(c) > 0$ for $c \in \mathcal{V}^{n-1}$. [**HINT:** The solution for α should be a function of w and c.]

Ans:
Let, count(w,c) denote the count of word w, found with/next to context c in N examples.

Now, $\sum_{i=1}^{N} \log(\alpha[w,c])$ categorically distributed over N sequences can be expressed as,

$$\sum_{i=1}^{N} \log(\alpha[w,c]) = count(w,c) \log(\alpha[w,c])$$

From (2) we have Lagrangian function, we insert the above for N sequences,

$$\mathcal{L}(\alpha,\lambda) = \sum_{w\in\mathcal{V};c\in\mathcal{V}^{n-1}} count(w,c)\ \log(\alpha[w,c]) - \lambda(\sum_{w\in\mathcal{V};c\in\mathcal{V}^{n-1}} \alpha[w,c] - 1)$$

Solving the Lagrangian function:

$$max\ \mathcal{L}(\alpha,\lambda) = \frac{d}{d\alpha}\mathcal{L}(\alpha,\lambda) = 0$$

Now, expanding Lagrangian and taking derivative for any (w,c),

$$\frac{d}{d\alpha}\mathcal{L}(\alpha,\lambda) = \frac{d}{d\alpha}[count(w,c)\ \log(\alpha[w,c]) - \lambda(\alpha[w,c]-1)]$$

$$= \frac{count(w,c)}{\alpha[w,c]} - \lambda$$

Since, $max\ \mathcal{L}(\alpha,\lambda) = \frac{d}{d\alpha}\mathcal{L}(\alpha,\lambda) = 0$,

$$\frac{count(w,c)}{\alpha[w,c]} - \lambda = 0 \implies \alpha[w,c] = \frac{count(w,c)}{\lambda} \qquad [Eq\ (A)]$$

From constraint $g(\alpha)$ in (1) and (2), inserting above in equation,

$$\sum_{w\in\mathcal{V};c\in\mathcal{V}^{n-1}} \alpha[w,c] - 1 = 0 \implies \sum_{w\in\mathcal{V};c\in\mathcal{V}^{n-1}} \frac{count(w,c)}{\lambda} = 1 \implies \lambda = \sum_{w\in\mathcal{V};c\in\mathcal{V}^{n-1}} count(w,c)$$

Using this in Eq(A), we get,

$$\alpha[w,c] = \frac{count(w,c)}{\sum_{w\in\mathcal{V};c\in\mathcal{V}^{n-1}} count(w,c)}$$

Problem 2: Noise Contrastive Estimation

In this problem, we will explore efficient training of neural language models using noise-contrastive estimation. Recall that in neural language modeling, the conditional probability $p(w \mid c)$ is modeled by

$$p_\theta(w \mid c) = \frac{\exp(f_\theta(w, c))}{\sum_{w' \in \mathcal{V}} \exp(f_\theta(w', c))} \ , \tag{1}$$

where $w \in \mathcal{V}$ is a token in the vocabulary, $c \in \mathcal{C}$ is some context, and $f_\theta \colon \mathcal{V} \times \mathcal{C} \to \mathbb{R}$ is a scoring function indicating how compatible w and c are, e.g. a recurrent neural network.

1. [2 points] As usual, we use MLE to learn the parameters $\theta \in \mathbb{R}^d$. Show that the gradient of the log likelihood for a single observation $\ell(\theta, w)$ is

$$\nabla_\theta f_\theta(w, c) - \mathbb{E}_{w \sim p_\theta}[\nabla_\theta f_\theta(w, c)] \ .$$

Ans:

$$MLE = max\ \ell(\theta) = max\ \log(p_\theta(w \mid c)) = max\ \log[\frac{\exp(f_\theta(w, c))}{\sum_{w' \in \mathcal{V}} \exp(f_\theta(w', c))}]$$

$$= max\ [\log(\exp(f_\theta(w, c))) - log(\sum_{w' \in \mathcal{V}} \exp(f_\theta(w', c)))]$$

$$= max\ [f_\theta(w, c) - \sum_{w' \in \mathcal{V}} f_\theta(w', c)]$$

Now gradient of $\ell(\theta) = \frac{d\ell(\theta)}{d\theta}$,

$$\frac{d\ell(\theta)}{d\theta} = \frac{d}{d\theta}[f_\theta(w, c) - \sum_{w' \in \mathcal{V}} f_\theta(w', c)]$$

$$= \nabla_\theta f_\theta(w, c) - \sum_{w' \in \mathcal{V}} \nabla_\theta f_\theta(w', c)$$

Since, gradient of $f_\theta(w', c)$ summed over all $w' \in \mathcal{V}$, gives the expected value for $\nabla_\theta f_\theta(w, c)$ for word w in probability distribution of θ. Thus, the above expression can be written as below.

$$\frac{d\ell(\theta)}{d\theta} = \nabla_\theta f_\theta(w, c) - \mathbb{E}_{w \sim p_\theta}[\nabla_\theta f_\theta(w, c)]$$

2. [2 points] Note that computing the gradient can be expensive due to summing over the vocabulary when computing the expectation term, which arises from the normalizer (or the partition function) in (1). One idea is to treat the normalizer as another parameter to estimate, i.e.

$$p_\theta(w \mid c) = \frac{\exp(f_\theta(w, c))}{\exp(z_c)} ,$$

where $z_c \in \mathbb{R}$ for each context c. Explain why the MLE solution for z_c doesn't exist.

Ans:
As z_c is a summed up function on all possible (w,c) pairs, it can keep growing infinitely, which means the likelihood for z_c can be estimated to ∞. Thus, there is no fixed maximum that can be likely estimated. Hence, we conclude that MLE for z_c does not exist.

3. [3 points] The key idea in noise contrastive estimation is to reduce the density estimation problem to a binary classification problem, i.e. deciding whether a word comes from the "true" distribution $p(w \mid c)$ or a "noise" distribution $p_n(w)$. Note that the noise distribution is context-independent. (This should remind you of negative sampling in HW1.) Now consider a new data-generating process: Given context c, with probability $\frac{1}{k+1}$ we sample a word from $p(w \mid c)$; with probability $\frac{k}{k+1}$ we sample a word from $p_n(w)$ ($k \in \mathbb{N}$). In other words, for each "true" sample, we generate k "fake" samples. Let Y be a binary random variable indicating whether w is a true sample or a fake sample. Show that

$$p(Y = 1 \mid w, c) = \frac{p(w \mid c)}{p(w \mid c) + kp_n(w)} \ .$$

[**HINT:** Use Bayes' rule.]

Ans:
From Bayes' rule,

$$p(Y = 1 \mid w, c) = \frac{p(w, c \mid Y = 1)p(Y = 1)}{p(w, c)}$$

$$= \frac{p(true(w, c))}{p(w, c)} = \frac{p(true(w, c))}{p(true(w, c)) + p(false(w, c))}$$

Since each true value comes from $p(w \mid c)$ and false value comes from $p_n(w)$ with probabilities $\frac{1}{k+1}$ and $\frac{k}{k+1}$ respectively. We can rewrite the above expression as:

$$\frac{p(true(w, c))}{p(true(w, c)) + p(false(w, c))} = \frac{\frac{1}{k+1}p(w \mid c)}{\frac{1}{k+1}p(w \mid c) + \frac{k}{k+1}p_n(w)}$$

Simplifying the above, we get,

$$p(Y = 1 \mid w, c) = \frac{p(w \mid c)}{p(w \mid c) + kp_n(w)}$$

4. [4 points] **[Optional]** We have reduced the problem of estimating $p(w \mid c)$ to predicting whether a sample (w, c) is true or fake. To learn a classifier, let's parametrize $p(Y = 1 \mid w, c)$. Note that p_n is known since it's chosen by us, so we just need to parametrize $p(w \mid c)$. Recall that we do not want to compute the normalizer, so (1) is not an option. Instead, let's model the normalizer as another parameter. We can either explicitly model it as in (2), or directly learn a self-normalizing function (i.e. $z_c = 0$):

$$\tilde{p}_\theta(w \mid c) = \exp(g_\theta(w, c)) \ .$$

Here we will proceed with the latter.[1]

For each word, we sample k fake words w^n from $p_n(w)$. Thus the log likelihood for a word and its noise samples is

$$\ell_{\text{NCE}}(\theta, w, k) = \log p_\theta(Y = 1 \mid w, c) + k\mathbb{E}_{w' \sim p_n} \log p_\theta(Y = 0 \mid w', c) \ .$$

In practice, expectation over p_n is approxmiated by k Monte Carlo samples.

Next, let's analyze how this objective connects to the MLE objective. Let $p_D(w \mid c)$ be the true distribution of words and consider the expected log likelihood. Let θ^* be the solution of

$$\max_{\theta \in \mathbb{R}^d} \mathbb{E}_{w \sim p_D} \left[\ell_{\text{MLE}}(\theta, w)\right]$$

and θ_n^* be the solution of

$$\max_{\theta \in \mathbb{R}^d} \mathbb{E}_{w \sim p_D} \left[\ell_{\text{NCE}}(\theta, w, k)\right] \ .$$

Assuming f_θ and g_θ have the same parametrization, show that when $k \to \infty$, θ^* and θ_n^* satisfy the same first order condition, i.e.

$$\mathbb{E}_{w \sim p_D} \left[\nabla_\theta f_\theta(w, c)|_{\theta=\theta^*}\right] = \mathbb{E}_{w \sim p_{\theta^*}} \left[\nabla_\theta f_\theta(w, c)|_{\theta=\theta^*}\right] \ ,$$

$$\mathbb{E}_{w \sim p_D} \left[\nabla_\theta g_\theta(w, c)|_{\theta=\theta_n^*}\right] = \mathbb{E}_{w \sim \tilde{p}_{\theta_n^*}} \left[\nabla_\theta g_\theta(w, c)|_{\theta=\theta_n^*}\right] \ .$$

Ans:
Using $\frac{d\ell_{MLE}(\theta,w)}{d\theta}$ from (1),

$$\max_{\theta \in \mathbb{R}^d} \mathbb{E}_{w \sim p_D} \left[\ell_{\text{MLE}}(\theta, w)\right] \implies \mathbb{E}_{w \sim p_D} \left[\frac{d\ell_{MLE}(\theta, w)}{d\theta}\right] = 0$$

$$\implies \mathbb{E}_{w \sim p_D} \left[\nabla_\theta f_\theta(w, c) - \mathbb{E}_{w \sim p_\theta}[\nabla_\theta f_\theta(w, c)]\right] = 0$$

For $\theta = \theta^*$

$$\mathbb{E}_{w \sim p_D} \left[\nabla_\theta f_\theta(w, c)|_{\theta=\theta^*}\right] - \mathbb{E}_{w \sim p_{\theta^*}} \left[\nabla_\theta f_\theta(w, c)|_{\theta=\theta^*}\right] = 0$$

Hence, we conclude,

$$\mathbb{E}_{w \sim p_D} \left[\nabla_\theta f_\theta(w, c)|_{\theta=\theta^*}\right] = \mathbb{E}_{w \sim p_{\theta^*}} \left[\nabla_\theta f_\theta(w, c)|_{\theta=\theta^*}\right]$$

Similarly,

$$\max_{\theta \in \mathbb{R}^d} \mathbb{E}_{w \sim p_D} \left[\ell_{\text{NCE}}(\theta, w, k)\right] \implies \mathbb{E}_{w \sim p_D} \left[\frac{d\ell_{NCE}(\theta, w)}{d\theta}\right] = 0$$

As,

$$\ell_{\text{NCE}}(\theta, w, k) = \log p_\theta(Y = 1 \mid w, c) + k\mathbb{E}_{w' \sim p_n} \log p_\theta(Y = 0 \mid w', c)$$

[1] Empirically, it has been observed that setting z_c to be a constant works just fine when g_θ is a neural network.

$\frac{d\ell_{NCE}(\theta,w)}{d\theta}$ can be derived as:

$$\frac{d\ell_{NCE}(\theta,w)}{d\theta} = \frac{d}{d\theta}\left[\log p_\theta(Y=1 \mid w,c) + k\mathbb{E}_{w'\sim p_n}\log p_\theta(Y=0 \mid w',c)\right]$$

$$= \frac{d}{d\theta}\left[\log p_\theta(Y=1 \mid w,c)\right] + k\mathbb{E}_{w'\sim p_n}\frac{d}{d\theta}\left[\log p_\theta(Y=0 \mid w',c)\right] \qquad [Eq\ (B)]$$

Now, from (3),

$$\frac{d}{d\theta}\left[\log p_\theta(Y=1 \mid w,c)\right] = \frac{d}{d\theta}\log\left[\frac{p(w \mid c)}{p(w \mid c)+kp_n(w)}\right]$$

$$= -\frac{d}{d\theta}\log\left[\frac{p(w \mid c)+kp_n(w)}{p(w \mid c)}\right] = -\frac{d}{d\theta}\log\left[1+\frac{kp_n(w)}{p(w \mid c)}\right]$$

Since, $\log\ \tilde{p}_\theta(w \mid c) = \log\ \exp(g_\theta(w,c)) = g_\theta(w,c)$

$$= \frac{kp_n(w)}{p(w \mid c + kp_n(w)}\frac{d}{d(\theta)}\log p(w \mid c) = \frac{kp_n(w)}{p(w \mid c + kp_n(w)}\nabla_\theta g_\theta(w,c)$$

Similarly, we find the derivative for 2nd term in Eq (B),

$$\frac{d}{d\theta}\left[\log p_\theta(Y=0 \mid w',c)\right] = -\frac{p(w \mid c)}{p(w \mid c)+kp_n(w)}\nabla_\theta g_\theta(w,c)$$

Inserting the above results in Eq (B) to derive max for NCE, for $\theta = \theta_n^*$ we get,

$$\mathbb{E}_{w\sim p_D}\left[\frac{d\ell_{NCE}(\theta,w)}{d\theta}\right] = \mathbb{E}_{w\sim p_D}\left[\frac{kp_n(w)}{p(w \mid c + kp_n(w)}\nabla_\theta g_\theta(w,c)\right] + k\mathbb{E}_{w'\sim p_n}\left[-\frac{p(w \mid c)}{p(w \mid c)+kp_n(w)}\nabla_\theta g_\theta(w,c)\right]$$

$$\implies \mathbb{E}_{w\sim p_D}\left[\frac{kp_n(w)}{p(w \mid c + kp_n(w)}\nabla_\theta g_\theta(w,c)\right] - \mathbb{E}_{w'\sim p_n}\left[\frac{kp(w \mid c)}{p(w \mid c)+kp_n(w)}\nabla_\theta g_\theta(w,c)\right] = 0$$

When $k \to \infty$ the p terms in fraction can be ignored, thus for $\theta = \theta_n^*$, we get,

$$\mathbb{E}_{w\sim p_D}\left[\nabla_\theta g_\theta(w,c)|_{\theta=\theta_n^*}\right] - \mathbb{E}_{w\sim \tilde{p}_{\theta_n^*}}\left[\nabla_\theta g_\theta(w,c)|_{\theta=\theta_n^*}\right] = 0$$

Hence, we conclude,

$$\mathbb{E}_{w\sim p_D}\left[\nabla_\theta g_\theta(w,c)|_{\theta=\theta_n^*}\right] = \mathbb{E}_{w\sim \tilde{p}_{\theta_n^*}}\left[\nabla_\theta g_\theta(w,c)|_{\theta=\theta_n^*}\right]$$

Problem 3: Conditional Random Fields

In this problem, you will implement inference algorithms for the CRF model and compare different sequence prediction models on synthetic data. You may want to go over the `mxnet_tutorial.ipynb` first before you start.

Environment setup: Follow instructions in `README.md` to set up the environment for running the code.

(a) [2 points] To get started, take a look at the function `generate_dataset_identity` in `util.py` and the class `UnigramModel` in `model.py`. Given $x = (x_1, \dots x_n)$ where $x_i \in \mathcal{V}$, the model makes an independent prediction at each step using only input at that step, i.e. $p(y_i \mid x_i)$. Run `python test.py unigram` to train a `UnigramModel`. It outputs the average hamming loss in the end. Let $y = (y_1, \dots, y_n)$ be the gold labels and $\hat{y} = (\hat{y}_1, \dots, \hat{y}_n)$ be the predicted labels, take a look at `hamming_loss` in `submission.py` and write down the loss function.

Ans:
Loss function can be represented as below:

$$Loss = 1 - \frac{count(\ success\ )}{count(\ success\ +\ failure)}$$

$$\implies Loss = \frac{\sum_{j=1}^{m} \sum_{y \in \mathcal{V}; \hat{y} \in \mathcal{V}} \mathbb{1}_{y_i^{(j)} \neq \hat{y}_i^{((j)}}}{m * |\mathcal{V}|}$$

where, m is the sample size and n is the number of labels in each sample.

(b) [2 points] Take a look at the `RNNModel` in `model.py`. It uses a bi-directional LSTM to encode x and makes independent predictions for each y_i. This time let's use the dataset generated by `generate_dataset_rnn`. Compare the result by running `python test.py unigram --data rnn` and `python test.py rnn --data rnn`. Which model has a lower error rate? Explain your findings.

Ans:
Unigram model has much higher error rate as compared to RNN model. Since unigram model has weight only based on each input, whereas RNN model, using bi-directional LSTM to encode x, weighs its predictions based on past and future computations. Unigram model treats each label as independent of next and previous labels. Also, the dataset we are using has reverse order of sequence of x_i and y_i. Hence, Unigram was thus unable to predict correct sequences when the sequence of the input is reversed, whereas RNN was able to do so.

(c) [4 points, coding] Next, we are going to add a CRF layer on top of the RNN model (see `CRFRNNModel` in `model.py`). Here we use the autograd function in MXNet to compute gradient for us, so we only need to implement the forward pass (the counterpart of the forward algorithm). Take a look at `crf_loss`. The main challenge here is to compute the normalizer which sums over all possible sequences:

$$\begin{aligned}\text{normalizer} &= \sum_{y \in \mathcal{Y}^n} \exp\left[s(y)\right] \\ &= \sum_{y \in \mathcal{Y}^n} \exp\left[\sum_{i=1}^{n} u(y_i) + \sum_{i=2}^{n} b(y_i, y_{i-1})\right]\end{aligned}$$

where u and b are scores from the `CRFRNNModel`. Note that here we assume $y_1 = *$ (the start symbol). Implement `compute_normalizer` using the `logsumexp` function in `util.py`. Your result must match `bruteforce_normalizer`. [**HINT:** You can compute all sums using array operations. `np.expand_dims` is very helpful here.]

See `submission.py`. No written submission.

(d) [4 points, coding] During inference, we will use Viterbi decoding to find

$$\arg\max_{y \in \mathcal{Y}^n} s(y)$$

where $s(y) = \sum_{i=1}^{n} u(y_i) + \sum_{i=2}^{n} b(y_i, y_{i-1})$. Implement `viterbi_decode`. Your result must match `bruteforce_decode`. [**HINT:** You can compute all sums using array operations. `np.expand_dims` is very helpful here.]

See `submission.py`. No written submission.

(e) [3 points] We are ready to test the CRFRNN model now. Use the HMM data (take a look at `generate_dataset_hmm` in util.py) and compare it with the RNN model by running `python test.py rnn --data hmm` and `python test.py crfrnn --data hmm`. Compare the results. [**NOTE:** This is an open-ended question. Discuss any findings you have is fine, e.g. runtime, error rate, convergence rate etc.]

Ans:
CRF-RNN has much higher runtime as compared to RNN. This is due to the computation of normalizer and decoder. Which is why it is essential to use dynamic programming / Viterbi algorithm which reduces the runtime manifolds as compared to the bruteforce implementation of the same. As observed, RNN took 7.16 seconds to run, whereas bruteforce implementation of CRF never completed even after hours. With Dynamic programming, the runtime of CRF-RNN was 111.74 seconds to run. The error rates were nearly 0.20 for both RNN and CRF-RNN, with CRF-RNN having slightly better error rates (2-3%). Convergence rate for RNN was better as compared to CRF-RNN. Overall, in my opinion, with only slight improvement in error rates and much higher runtime, I would prefer RNN over CRF-RNN unless even slight accuracy improvement is of vital importance.